Algorithms for Arbitrary Precision Floating Point Arithmetic*

Douglas M. Priest
Department of Mathematics
University of California, Berkeley

March 19, 1991

Abstract

We present techniques which may be used to perform computations of very high accuracy using only straightforward floating point arithmetic operations of limited precision, and we prove the validity of these techniques under very general hypotheses satisfied by most implementations of floating point arithmetic.

To illustrate the application of these techniques, we present an algorithm which computes the intersection of a line and a line segment. The algorithm is guaranteed to correctly decide whether an intersection exists and, if so, to produce the coordinates of the intersection point accurate to full precision. Moreover, the algorithm is usually quite efficient; only in a few cases does guaranteed accuracy necessitate an expensive computation.

1. Introduction

"How accurate is a computed result if each intermediate quantity is computed using floating point arithmetic of a given precision?" The casual reader of Wilkinson's famous treatise [21] and similar roundoff error analyses might conclude that the most one can hope to say about the accuracy of a computation carried out in fixed precision floating point arithmetic is that the computed solution is close to the exact solution of a problem close to the given problem, where the precise meaning of "close" depends on the precision of the arithmetic. He might further surmise that if he wishes to compute a result with more accuracy than such an analysis can guarantee based on the widest precision of arithmetic supported in whatever computing environment is available, then he must instead resort to a subroutine library such as Brent's MP package [3] in order to compute with higher precision arithmetic.

That both conclusions are wrong follows from the existence of techniques which allow a program to compute to arbitrarily high accuracy using only fixed precision floating point arithmetic operations. These techniques can be used in virtually any computing

*Originally published in *Proc. 10th Symposium on Computer Arithmetic*, P. Kornerup and D. Matula, Eds., IEEE Computer Society Press, Los Alamitos, Calif., 1991

environment which incorporates a "reasonable" floating point arithmetic, and they do not rely on dirty tricks such as accessing a floating point number as though it were an integer value, nor on inefficient conversions such as extracting the exponent field of a floating point number. Instead they use only floating point additions, comparisons, multiplications, and divisions, which are commonly supported in hardware and often greatly optimized. Armed with such techniques, a programmer can write code which performs none but straightforward floating point operations, looks entirely ordinary both to the eye and to the compiler, runs almost as efficiently as any other floating point computation, and produces a result whose accuracy is limited only by the overflow and underflow threshholds.

Of course, extra accuracy is not free: a program guaranteed to produce an accurate answer must be more expensive than one which is allowed to emit erroneous results. How much more expensive must such a program be? In this paper, we present algorithms which suggest that the cost of extra accuracy may be quite reasonable for many problems—at least, reasonable enough that we are obliged to consider more carefully the trade-offs between cost and accuracy. Our algorithms are based on an approach pioneered by Møller [17], Kahan [7], Dekker [4], Pichat [19], Linnainmaa [13, 14], and several others [10, 12], but our hypotheses are slightly more general than theirs. Moreover, rather than simply extending the accuracy to approximately twice the working precision, as do most of the aforementioned references, our algorithms expand upon methods developed by Bohlender [1] and Kahan [9] which compute to arbitrarily high accuracy.

Below, we give algorithms for exact addition and multiplication and arbitrarily accurate division of extended precision numbers using only fixed precision floating point arithmetic operations. We express the cost of these algorithms in terms of the number of fixed precision operations required. Section 2 describes floating point arithmetics and defines the criteria which an arithmetic must satisfy for our results to be valid. Lemma 1 in this section generalizes results from [4, 9, 10, 13, 17, 19]. Sections 3 and 4 focus on addition algorithms for extended precision numbers: section 3 considers the exact addition of two such numbers, while section 4 presents an algorithm for the problem Kahan [9] has called "distillation", namely, expressing the exact sum of n arbitrary fixed precision floating point numbers as a single extended precision number. Kahan gives an algorithm which requires at most $O(n \log n)$ fixed precision operations; we present another, very different algorithm which nevertheless has the same cost bound. Sections 5 and 6 present algorithms for multiplication and division, respectively. Although the techniques motivating these algorithms are well known, we present the algorithms both for completeness and to obtain explicit cost bounds. Section 7 gives an example of an algorithm which uses several of our extended precision arithmetic routines to compute to full accuracy the point of intersection of a line and a line segment. Finally, section 8 summarizes our results and discusses some related topics.

With the exception of section 7 and several remarks in section 8, we ignore the possibility of overflow and underflow. Therefore, unless otherwise noted, the phrase "provided no overflow or underflow occurs" should be appended to each result stated below. We stress, however, that we are not advocating a complete, self-contained software package upon which a user must rely if he wishes to perform extended precision arithmetic, but rather a paradigm for numerical computation in which each intermediate quantity is computed to such precision as is necessary to guarantee the desired accuracy in the final result. Our intent is merely to demonstrate the feasibility of such a paradigm up to the limits imposed by overflow and underflow; in most cases, those limits are not particularly constraining, as our example in section 7 illustrates.

2. Properties of Floating Point Arithmetics

We begin by defining floating point arithmetic and noting several important properties. For integers t and β both greater than 1, let $R_{\beta,t}$ denote the set of all rational numbers of the form $m\beta^k$ where m and k are integers and $|m| < \beta^t$. These numbers are called the *t-digit, radix β floating point numbers.* A *floating point arithmetic* of radix β and t-digit precision is one which, given any $a, b \in R_{\beta,t}$ and $\circ \in \{+, -, \times, /\}$, determines a quantity $c \in R_{\beta,t}$ (assuming $b \neq 0$ if $\circ = /$, and subject to the usual caveats regarding overflow and underflow). We write $c = \mathrm{fl}(a \circ b)$ to denote the result of computing $a \circ b$ in floating point arithmetic.

We view arbitrary precision arithmetic in the following context. Let R_β denote the subring of $\mathbf{Q}$ generated by $R_{\beta,t}$. Note that each $x \in R_\beta$ may be written as a finite sum $x = \sum x_i$ with $x_i \in R_{\beta,t}$. In particular, if $x \neq 0$ we can choose the x_i satisfying a *non-overlapping condition*: writing $x_i = m_i\beta^{k_i}$ with $\beta^{t-1} \leq |m_i| < \beta^t$, we require that $k_i - k_j \geq t$ for $i < j$. (In other words, if $i < j$ then the most significant digit of x_j is at least one order of magnitude smaller than the least significant digit of x_i.) Such an expression is called a *t-digit expansion* for x; each term of the sum is a *component* of the expansion. A t-digit expansion for zero consists of a single zero component. Our goal is to show that all arithmetic operations over the ring R_β may be computed using only t-digit floating point arithmetic on components of t-digit expansions. (Here the "arithmetic" operations over R_β consist of the usual addition and multiplication as well as an approximate division which produces a quotient as an expansion accurate to a specified number of components.)

To guarantee that we can reduce arithmetic over R_β to fixed precision floating point arithmetic, we must make some assumptions about the nature of the floating point arithmetic we are using. The most important assumption we shall make is embodied in the following definition.

Definition: For t-digit numbers a and b and $\circ \in \{+, -, \times, /\}$, let $c = a \circ b$ exactly (assuming $b \neq 0$ if $\circ = /$). Suppose x and y are consecutive t-digit floating point numbers with the same sign as c such that $|x| \leq |c| < |y|$. Then the floating point arithmetic is called *faithful* if $\mathrm{fl}(a \circ b) = x$ whenever $c = x$ and $\mathrm{fl}(a \circ b)$ is either x or y whenever $c \neq x$.

All of the results which follow assume that the floating point arithmetic is faithful. In fact, most results assume nothing else; only the multiplication and division algorithms of sections 5 and 6 require additional hypotheses in the form of modest lower limits on the precision of the arithmetic. As we shall see, faithfulness is a powerful property, yet our assumption does not severely limit the applicability of our algorithms. Any arithmetic which conforms to the IEEE 754 or 854 standard is faithful, as are DEC VAX and IBM 370 arithmetics. In fact, any arithmetic in which subtraction is performed using at least one guard digit possesses faithful addition; faithful multiplication and division are not much more difficult to achieve. (In fairness, however, we must warn the reader that a number of machines still lack faithful arithmetic: notable exceptions include Crays and CDC Cybers. Of course, with somewhat more complicated algorithms and substantially more complicated proofs, our results could be extended to those machines as well.)

We now prove a crucial lemma which says that we can compute exactly the error incurred in the fixed precision addition of two t-digit numbers. This technique forms the basis of the algorithms presented in subsequent sections. As we will note below, the proof given here generalizes various special cases which are considered in [4, 9, 10, 13, 17, 19]. In addition to the hypothesis of faithfulness, we rely heavily on the following well-known property of floating point numbers (see Sterbenz [20]): if a and b are t-digit floating point numbers such that $1/2 \leq a/b \leq 2$ then $a - b$ is also a t-digit floating point number. In particular, if the floating point arithmetic is faithful, $\mathrm{fl}(a - b) = a - b$ exactly.

Lemma 1: Let a and b be any t-digit numbers. If the floating point arithmetic is faithful then the t-digit numbers c and d calculated by the following algorithm satisfy $c + d = a + b$ and either $d = 0$ or $c + d$ is an expansion for $a + b$ (i.e., $c = m_c\beta^{k_c}$ and $d = m_d\beta^{k_d}$ with $\beta^{t-1} \leq |m_c|, |m_d| < \beta^t$ and $k_c - k_d \geq t$).

Algorithm 1: (Addition of two t-digit numbers with explicit error term)

```
procedure sum_err( a, b )
begin
if |a| < |b|
    swap( a, b )
c := fl(a + b), e := fl(c − a)
g := fl(c − e), h := fl(g − a), f := fl(b − h)
d := fl(f − e)
if fl(d + e) ≠ f
```

```
      c := a, d := b
return c, d
end
```

Proof: Without loss, assume $|a| \geq |b|$. If $b = 0$ then clearly the algorithm produces $c = a$, $d = 0$, so the lemma holds. For the general case, assume $a > 0$; the proof when $a < 0$ is essentially identical. Let $c = \mathrm{fl}(a + b)$ as above and define $r = a + b - c$, so r is the error in computing c. We prove the lemma by establishing three claims.

Claim 1: $d = r$ if and only if r is a t-digit floating point number. Of course, if r is not a t-digit number, then we cannot have $d = r$. For the converse, suppose first that $-a \leq b \leq -a/2$. Then $1 \leq a/(-b) \leq 2$ so $a + b = a - (-b)$ is a t-digit number; hence, by faithfulness, $c = \mathrm{fl}(a + b) = a + b$ exactly and $r = 0$. Again by faithfulness, we must have $e = \mathrm{fl}(c - a) = b$, $g = a$, $h = 0$, $f = b$, and $d = 0 = r$, a t-digit number.

Now suppose $-a/2 < b < 0$. Let a' denote the largest t-digit number not greater than $a/2$, and note that $a - a'$ is the smallest t-digit number not less than $a/2$. Since $|b| \leq a'$, we must have $a + b \geq a - a'$, so by faithfulness, $c = \mathrm{fl}(a + b) \geq a - a' \geq a/2$. But then $1/2 \leq c/a \leq 1$, so $e = \mathrm{fl}(c - a) = c - a = b - r$ exactly. Again we have $g = a$, $h = 0$, and $f = b$, so if r is a t-digit number then $d = \mathrm{fl}(f - e) = r$.

Finally, suppose $b > 0$. If $c \leq 2a$ then $1 \leq c/a \leq 2$ so $e = \mathrm{fl}(c - a) = c - a = b - r$ exactly. As before, $g = a$, $h = 0$, $f = b$, and thus $d = r$ if r is a t-digit number. If instead $c > 2a$, then $e = \mathrm{fl}(c - a)$ may not be computed exactly; let $s = c - a - e$, so s is the roundoff error incurred in computing e. By arguments similar to those of the preceding paragraph, we have $e \geq c/2$, so $1 \leq c/e \leq 2$ and $g = \mathrm{fl}(c - e) = c - e = a + s$ exactly. Now write $a = m\beta^k$ and $b = n\beta^j$ with $\beta^{t-1} \leq |m| < \beta^t$ and likewise for n. We can obtain $c > 2a$ only when $j = k$, so all computed quantities must be integer multiples of β^k. Likewise r is a multiple of β^k, but we must have $|r| < \beta^{k+1}$, so r is a t-digit number. (In fact, r is a one-digit multiple of β^k.) We also see that $|s| < \beta^{k+1}$, hence $h = \mathrm{fl}(g - a) = s$ exactly, and clearly $f = \mathrm{fl}(b - h) = b - s$ is also exact. Thus $d = \mathrm{fl}(f - e) = r$, and the claim holds.

Claim 2: $\mathrm{fl}(d + e)$ is always computed exactly. Recall that $\mathrm{fl}(d + e) = f$ exactly whenever r is a t-digit number. Suppose r is not a t-digit number. Then we must have $e = \mathrm{fl}(c - a) = c - a = b - r$, $g = a$, $h = 0$ and $f = b$, so $d = \mathrm{fl}(b - e)$. Write $b = n\beta^j$ with $\beta^{t-1} \leq |n| < \beta^t$ and note that a, b, and c are all integer multiples of β^j, so r is an integer multiple of β^j. If $br \geq 0$ then clearly $|r| \leq |b|$, so r would be a t-digit number; therefore suppose $br < 0$ and also $|r| > |b|$. Then $de < 0$ and $1/2 \leq d/(-e) \leq 1$ so $\mathrm{fl}(d + e) = \mathrm{fl}(d - (-e)) = d + e$ exactly (although we need not have $d + e = f$).

Claim 3: If r is not a t-digit number then a and b satisfy the non-overlapping condition. To see this, assume that r is not a t-digit number, and write $c = m\beta^k$, $a = n\beta^j$ with $\beta^{t-1} \leq m < \beta^t$ and likewise for n; we must show $|b| < \beta^j$. Suppose $|b| \geq \beta^j$ and recall that $d \neq r$ only if $|b| < |r|$. By faithfulness, $|r| < \beta^k$, so we must have $j < k$, but since $|b| \leq |a|$ this can happen only if $j = k - 1$. Now a, b, and c are all integer multiples of $\beta^{j-t+1} = \beta^{k-t}$, so r is an integer multiple of β^{k-t} and $|r| < \beta^k$. Thus r is a t-digit number, a contradiction.

From the preceding claims, the proof of the lemma proceeds as follows. If r is a t-digit number then $d = r$ and the condition in line 8 must fail because $\mathrm{fl}(d + e) = f$ exactly; in this case, the algorithm outputs the computed sum and the roundoff error which, by faithfulness, must satisfy the non-overlapping condition. Otherwise, the condition in line 8 must succeed because $\mathrm{fl}(d + e)$ is computed exactly but $d + e \neq f$, so the algorithm simply returns a and b; but in this case a and b already satisfy the non-overlapping condition. ∎

By examining the preceding proof, we can easily obtain several corollaries which show that the algorithm may be simplified substantially when the floating point arithmetic possesses additional properties besides faithfulness. Many of these special cases have been considered separately in some of the references; to clarify the relationship to our results, we introduce several definitions. Let a, b be any t-digit floating point numbers and $\circ$ one of $\{+, -, \times, /\}$ (with $b \neq 0$ if $\circ = /$). Let $c = a \circ b$ and x and y two consecutive t-digit numbers with $|x| \leq |c| < |y|$. Then the floating point arithmetic is *correctly rounding* if it is faithful and also $\mathrm{fl}(a \circ b) = x$ whenever $|c - x| < |c - y|$ and $\mathrm{fl}(a \circ b) = y$ whenever $|c - y| < |c - x|$. The arithmetic's addition is *properly truncating* if $\mathrm{fl}(a + b) = x$ whenever $c = x$ or $ab > 0$, but $\mathrm{fl}(a + b) = y$ whenever $c \neq x$ and $ab < 0$. We have borrowed the last definition from Dekker [4]; the reader should not confuse this term, as Linnainmaa did [13], with the notion of *correctly chopping*, in which $\mathrm{fl}(a + b) = x$ always.

The following simplification was established by Knuth [10] for correctly rounding arithmetic; here we prove a slightly more general result which includes properly truncating arithmetics. (Møller [17] obtained a similar but weaker result using still different hypotheses.)

COROLLARY 1: If the arithmetic has the property that the roundoff error of a sum is always a t-digit number (as it is in correctly rounding and properly truncating arithmetics), then lines 8 and 9 may be eliminated from the above algorithm.

PROOF: Returning to the proof of the lemma, we are assuming that r is always a t-digit number; hence the test in line 8 never succeeds. ∎

Dekker [4] and Linnainmaa [13] consider a different variation, which also appears in Pichat [19]. Again, we state a slightly more general result which follows easily from the preceding lemma.

COROLLARY 2: If the arithmetic has the property that $|\mathrm{fl}(a + b)| \leq 2|a|$ for all $|b| < |a|$ (as it is in properly truncating arithmetics, correctly chopping arithmetics, and all faithful binary arithmetics), then line 6 may be eliminated and b substituted for f in lines 7 and 8 of the above algorithm. In particular, if the arithmetic also has the property of the preceding corollary, that the roundoff error of a sum is a t-digit number (e.g., in a properly truncating arithmetic or a binary correctly rounding arithmetic), then lines 6, 8, and 9 may all be eliminated and b substituted for f in line 7.

PROOF: For the first statement, note that the hypothesis implies $|c| \leq 2|a|$, so we always have $g = a$, $h = 0$, and $f = b$. The second statement follows by combining the first with the previous corollary. ∎

Although the original algorithm is an interesting theoretical result, in practice virtually every computer which provides a faithful floating point arithmetic can take advantage of one of the simplifications described in the corollaries. For example, arithmetic which conforms to the IEEE 754 standard is binary and correctly rounding, so the simplest version suggested in corollary 2 may be used. For different reasons, the same version may be used on DEC VAX, IBM 370, and similar computers. The only faithful arithmetics which arise in practice and for which neither simplification applies are decimal correctly rounding arithmetics found on some hand calculators.

In the results which follow, we state the cost of our algorithms as the maximum number of t-digit floating point arithmetic operations performed; note that the number of operations required in the preceding algorithms is at most eleven, but may be reduced to as few as six if a simplified version may be used. (Here and below we count a comparison of two numbers as one floating point operation, since comparison implies subtraction. We also count the absolute value operation as one t-digit floating point operation, namely a comparison against zero. One may regard this as pessimistic, since the sign of a floating point number can be, and often is, represented in one bit. On the other hand, separate counts for each type of operation may easily be obtained from the algorithms presented; we have chosen to combine the counts of different operations only for simplicity.)

3. Addition

The addition algorithm we present is a simple variation of the classical algorithm in which the radix points are aligned and digits are summed pairwise from right to left. The difference is that we are summing components rather than digits, and we regard the error of the computed sum as the "sum" of the components and the computed sum itself as a "carry". Since we can only add two components at once, we always add the carry of the last addition to the next smaller component not yet added. (In effect, we are merge-sorting the components of the two expansions by increasing magnitude and adding in this order.) The exact sum is then the sum of the final carry and the error terms from each preceding addition. Since the error terms need not satisfy the non-overlapping condition for expansions, we must renormalize their sum to obtain the result as a t-digit expansion.

We present the addition and renormalization algorithms separately since we will reuse the renormalization process later. To facilitate the proofs we need one more definition: for $0 \le d < t$, a finite sequence $x_1, x_2, \ldots, x_n$ of t-digit floating point numbers is said to *overlap by at most d digits* if for each $j = 1, 2, \ldots, n-1$ there exist $i \le j$ and k such that $x_1, \ldots, x_{i-1}$ are integer multiples of β^k, x_i is an integer multiple of β^{k-d}, $x_{i+1}, \ldots, x_j$ are all zero, and $|x_{j+1}| < \beta^k$. (Loosely speaking, this condition says that if the x_i were written in positional notation, the significant digits of any two successive non-zero terms would coincide in at most d digit positions, and moreover, no three terms would mutually coincide in any position. For example, in four digit decimal floating point, the sequence $12340, 5678, 9.123$ overlaps by at most three digits, but the sequence $12340, 5678, 91.23$ does not because all three numbers coincide in the tens place.) Note in particular that if $\beta^{k-1} \le |x_1| < \beta^k$ and the sequence $x_1, \ldots, x_n$ overlaps by at most d digits then $|\sum x_j| < \beta^{k+1}$; that is, the sum can carry over to at most one larger place than the largest term. Note also that the components of a t-digit expansion do not overlap at all (i.e., they overlap by at most zero digits).

Proposition 2.1: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components. If the arithmetic is faithful then the following algorithm computes a sequence $e_1, \ldots, e_{n+m}$ which overlaps by at most one digit and satisfies $x + y = \sum e_j$.

Algorithm 2.1: (Addition)

```
procedure add( n, x_1, ..., x_n, m, y_1, ..., y_m )
begin
i := n, j := m
if |x_i| < |y_j|
      while i > 1 and |x_{i-1}| <= |y_j|
            e_{i+j} := x_i, i := i - 1
else if |x_i| > |y_j|
      while j > 1 and |y_{j-1}| <= |x_i|
            e_{i+j} := y_j, j := j - 1
a := x_i, b := y_j
while i > 1 or j > 1
      ( c, e_{i+j} ) := sum_err( a, b )
      a := c
      if i = 1 or ( j > 1 and |y_{j-1}| < |x_{i-1}| )
            b := y_{j-1}, j := j - 1
      else
            b := x_{i-1}, i := i - 1
```

(c, e_2) := sum_err(a, b)
$e_1 := c$
return $n+m, e_1, \ldots, e_{n+m}$
end

PROOF: Clearly $x + y = \sum_{j=1}^{n+m} e_j$. To prove that the e_j overlap by at most one digit, we proceed by induction on the number of times the algorithm calls sum_err. In particular, we show that after each execution of sum_err there exists k such that $|e_{i+j}| < \beta^k$ and $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ are all integer multiples of β^{k-1} with all but one being multiples of β^k. The proposition will follow easily.

First observe that prior to the first call to sum_err, we have $i = n$ or $j = m$ or both. If $i = n$ but $j < m$ then the final loop starts with $e_{n+j+1}, \ldots, e_{n+m}$ equal to the last $m - j$ components of y, which do not overlap by assumption. Since this case can occur only if $y \neq 0$, we may write $y_j = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then $|x_n| \geq |y_j|$, so $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_j$ are all integer multiples of β^u, while $|e_{n+j+1}| = |y_{j+1}| < \beta^u$. A similar argument applies if $i < n$ and $j = m$. Hence in any case we may begin the induction assuming that the algorithm has entered the final loop; in particular, if $i > 1$ then $|x_{i-1}| > |y_j|$, whereas if $j > 1$ then $|y_{j-1}| > |x_i|$.

To start the induction, let $a = x_i$, $b = y_j$, and (c, e_{i+j}) denote the result returned from the first call to sum_err. If $c = 0$ then $e_{i+j} = 0$ and the initial case holds. Otherwise write $c = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ and note that $|e_{i+j}| < \beta^v$. Suppose $|a| \geq |b|$; the case $|a| < |b|$ is similar. Write $a = x_i = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then $x_1, \ldots, x_{i-1}$ are multiples of β^{u+t}, and since $|y_{j-1}| > |x_i|$ then y_{j-1} is a multiple of β^u so $y_1, \ldots, y_{j-2}$ are also multiples of β^{u+t}. Since $u \geq v - 1$, we find that $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ are all multiples of β^{v-1} and all but y_{j-1} are multiples of $\beta^{v-1+t} \geq \beta^v$ so again the initial case holds. Thus the proposition is established for the first call to sum_err.

Now suppose we have returned from the end of the final loop. Then the current value of a is the previous value of c and the current value of b is one of $\{x_i, y_j\}$. Let c and e_{i+j} be the quantities calculated in this call to sum_err and note that $c = \sum_{k=i}^{n} x_k + \sum_{k=j}^{m} y_k - \sum_{k=i+j}^{n+m} e_k$. If $c = 0$ then $e_{i+j} = 0$ and we need only appeal to the induction hypothesis. Otherwise write $c = q\beta^v$ with $\beta^{t-1} \leq |q| < \beta^t$ and again note $|e_{i+j}| < \beta^v$. Assume $b = x_i$; the case $b = y_j$ is similar. Note that $|b| \leq |y_{j-1}|$ (otherwise we would have chosen $b = y_{j-1}$ in the previous iteration of the loop) and that $|b| \geq |y_j|$ (otherwise we would have chosen $b = x_i$ prior to the previous iteration). Write $b = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Then

$$\beta^{v+t-1} \leq |c| \leq \left|\sum_{k=i}^{n} x_k\right| + \left|\sum_{k=j}^{m} y_k\right| + \left|\sum_{k=i+j}^{n+m} e_k\right|$$

$$< \beta^{u+t} + \beta^{u+t} + \beta^{v+1}$$

where the bounds on the first two terms follow from the non-overlapping conditions on x and y and the inequality $|y_j| \leq |b| = |x_i|$, and the bound for the last term follows from the induction hypothesis. Consequently, $\beta^{v+t-1} < \beta^{u+t+1} + \beta^{v+1}$, so $v + t - 1 < \max(u + t + 2, v + 2)$. Since $t \geq 3$ we must have $v + t - 1 < u + t + 2$ so $v < u + 3$ and hence $v \leq u + 2$. Now $x_1, \ldots, x_{i-1}$ are multiples of β^{u+t} and thus of β^v. To complete the induction it suffices to show that y_{j-1} is a multiple of β^{v-1}; then $y_1, \ldots, y_{j-2}$ will be multiples of $\beta^{v-1+t} \geq \beta^v$. Write $y_{j-1} = r\beta^s$ with $\beta^{t-1} \leq |r| < \beta^t$ so that $|\sum_{k=j}^{m} y_k| < \beta^s$. Then the previous inequality becomes $\beta^{v+t-1} < \beta^{u+t} + \beta^s + \beta^{v+1}$. Since $v \leq u + 2$, if $s < u + t$ then we have $v + t - 1 < u + t + 1$ so $v < u + 2$ and hence $v \leq u + 1$, and since $|y_{j-1}| \geq |b|$ this implies y_{j-1} is a multiple of $\beta^u = \beta^{v-1}$ and the induction step follows. If instead $s \geq u + t$ then clearly $s \geq v - 1$, so again y_{j-1} is a multiple of β^{v-1}.

From the above argument, we clearly must have either $c, x_1, \ldots, x_{i-1}, y_1, \ldots, y_{j-1}$ all multiples of β^v or else exactly one of $\{x_{i-1}, y_{j-1}\}$ is a multiple of β^{v-1} but not of β^v. Suppose as above that y_{j-1} is not a multiple of β^v. Then clearly $|y_{j-1}| < |x_{i-1}|$ so the next iteration of the loop will choose $b = y_{j-1}$. Consequently, either all of the quantities $e_1, \ldots, e_{i+j-1}$ are multiples of β^v or there exists $l \leq i + j - 1$ such that $e_1, \ldots, e_{l-1}$ are multiples of β^v, e_l is a multiple of β^{v-1}, and $e_{l+1}, \ldots, e_{i+j-1}$ are zero. This completes the proof. ∎

Proposition 2.2: Let $e_1, \ldots, e_n$ overlap by at most $t - 2$ digits. If the arithmetic is faithful then the following algorithm computes an expansion $s = \sum_{j=1}^{m} s_j$ with $m \leq n$ such that $s = \sum e_j$.

Algorithm 2.2: (Renormalization)

```
procedure renorm( n, e_1, ..., e_n )
begin
c := e_n
for i := n-1, n-2, ..., 1
      ( c, f_{i+1} ) := sum_err( c, e_i )
f_1 := c
s_1 := f_1, k := 1
for j := 2, 3, ..., n
      ( c, d ) := sum_err( s_k, f_j )
      s_k := c
      if d ≠ 0
            l := k - 1, k := k + 1
            while l ≥ 1
                  ( c', d' ) := sum_err( s_l, s_{l+1} )
                  if d' = 0
```

$s_l := c'$, $l := l-1$, $k := k-1$
else
$l := 1$
$s_k := d$
return $k, s_1, \ldots, s_k$
end

PROOF: After the first for loop terminates, we clearly have $\sum f_j = \sum e_j$. Furthermore, we claim that the f_j do not overlap at all. We again argue by induction. Specifically, let c and f_{i+1} be the return values from a call to sum_err in the first loop. If $c = 0$ then $f_{i+1} = 0$ so the induction holds; otherwise write $c = q\beta^v$ with $\beta^{t-1} \le |q| < \beta^t$ so that $|f_{i+1}| < \beta^v$. Also write $e_i = p\beta^u$ with $\beta^{t-1} \le |p| < \beta^t$. Then since the e_j overlap by at most $t-2$ digits, we have $v \le u+1$. Consequently, $e_1, \ldots, e_{i-1}$ are all multiples of $\beta^{u+2} \ge \beta^v$, so $f_1, \ldots, f_i$ must be multiples of β^v while $|f_{i+1}| < \beta^v$. This establishes the claim.

Proceeding to the rest of the algorithm, we note that the output $s_1, \ldots, s_k$ clearly satisfies $\sum_{j=1}^k s_j = \sum_{j=1}^n f_j$. We need only show that the output qualifies as an expansion. Induct on k: let c and d be the return values from a call to sum_err in the outer for loop. If $d = 0$ there is nothing to prove, so assume $d \ne 0$. Then also $s_k = c \ne 0$ so we may write $s_k = q\beta^v$ with $\beta^{t-1} \le |q| < \beta^t$ and we have $|d| < \beta^v$. Since the f_i do not overlap, we also have $|d + \sum_{i>j} f_i| < \beta^v$. Therefore each subsequent call to sum_err in the outer for loop must yield $|c| \le \beta^v$ with equality only if c is rounded (so that again $d \ne 0$). If we always have $|c| < \beta^v$ then clearly $|s_i| < \beta^v$ for $i > k$ so the induction holds. If instead we obtain $|c| = \beta^v$ at some later step, then by the induction hypothesis, upon completion of the predicate of the outer if statement, we still find $s_1, \ldots, s_k$ (for the value of k then in effect) is an expansion. In addition, we then have $|d| < \beta^{v-t}$, so again the induction holds, and the proof is complete. ∎

It follows from the proof above that sum_err is never called from the innermost while loop more than $n-1$ times. An easy operation count yields the following.

COROLLARY: Let x and y be t-digit expansions having n and m components, respectively. Then the composition of algorithms 2.1 and 2.2 computes a t-digit expansion $s = x + y = \sum_{i=1}^k s_i$ with $k \le n+m$ using at most $49(n+m-1)+9$ t-digit floating point operations.

4. Distillation

In this section we consider the following problem: given any t-digit floating point numbers $x_1, x_2, \ldots, x_n$, compute a t-digit expansion y such that $y = \sum_{i=1}^n x_i$. Kahan has coined the term "distillation" to descibe the process by which the components of y may be obtained from the x_i.

The first algorithm to implement distillation was given by Pichat [19], who was only interested in obtaining the first component of y. Pichat's algorithm simply passes through each of the x_i in turn applying the basic sum_err procedure. (In fact, the first loop in our renormalization procedure above constitutes one pass of Pichat's algorithm.) Pichat showed that this method converges; i.e., that after a sufficient number of passes, one obtains the first component of y. Bohlender [1] then showed that Pichat's algorithm can be used to obtain the entire expansion for y with at most $n-1$ passes, and he added a stopping criterion for the case where only a given number of leading components of y are desired.

Many other algorithms which evaluate a sum or an inner product to working precision (i.e., yielding the first component of an expansion) have been developed; see Bohlender [2] for a survey of several approaches. Unfortunately, many of these techniques rely on special features, such as an extra wide fixed point accumulator, directed roundings, or interval arithmetic, which must be implemented in a combination of hardware and low-level software to be efficient. Other techniques, such as one proposed by Kulisch and Miranker [11], require direct access to the exponent and significand fields of floating point numbers, and obtaining these quantities, even when the programming environment provides a convenient way to do so, is much too time-consuming compared to the highly optimized and streamlined floating point additions and comparisons of Pichat's algorithm, at least for the modest values of n one typically encounters.

On the other hand, Pichat's algorithm is not the only way to implement distillation using only faithful floating point addition and comparison. Kahan [9] gives a distillation algorithm which first sorts the x_i by decreasing magnitude, then uses successive passes like Pichat's algorithm, but alternating in order: first smallest to largest, then largest to smallest. Whereas Pichat's algorithm requires $O(n^2)$ t-digit floating point operations in the worst case (i.e., $n-1$ passes through n numbers), Kahan claims his algorithm requires at most $O(n \log n)$ operations. We now present yet another algorithm which is easily seen to have the same worst-case bound. Our method, a straightforward divide-and-conquer approach using the algorithms developed above, recursively adds successive pairs of partially distilled sums in a binary tree-like reduction. A similar approach was proposed for some parallel computer architectures by Leuprecht and Oberaigner [12].

Proposition 3: Given t-digit floating point numbers $x_1, x_2, \ldots, x_n$. Then the following algorithm carried out with faithful arithmetic computes a t-digit expansion $y = \sum_{i=1}^{m} y_i$ with $m \leq n$ such that $y = \sum x_i$. The algorithm requires at most $49n\lceil \log n \rceil$ t-digit floating point operations.

Algorithm 3: (Distillation)

procedure distill($n, x_1, \ldots, x_n$)
begin

if $n = 1$

$\quad l := 1,\ w_1 := x_1$

else

$\quad m := \lfloor n/2 \rfloor$

$\quad (\ j, y_1, \ldots, y_j\ ) :=$ distill($m, x_1, \ldots, x_m$)

$\quad (\ k, z_1, \ldots, z_k\ ) :=$ distill($n - m, x_{m+1}, \ldots, x_n$)

$\quad (\ l, w_1, \ldots, w_l\ ) :=$ renorm(add($j, y_1, \ldots, y_j,\ k, z_1, \ldots, z_k$))

return $l, w_1, \ldots, w_l$

end

PROOF: That the output is an expansion for $\sum x_i$ is obvious. As for the cost, note that the number of components of a sum as computed by the renormalize and add algorithms does not exceed the sum of the numbers of components of the summands; hence for each level of the binary tree, the total number of components of the expansions added at that level is at most n. Therefore the cost of each level is at most $49n$, and since the number of such levels is clearly $\lceil \log n \rceil$, the total cost is at most $49n\lceil \log n \rceil$. ∎

Kahan's algorithm in [9], though very different from the algorithm above, also incurs a cost at most proportional to $n \log n$, presumably with a modest constant, as opposed to a worst case cost of $11n(n-1)$ for Pichat's algorithm (assuming the first version of the sum_err procedure is used). Although algorithms are known which have asymptotic cost at most proportional to n, they are impractical for most purposes since they explicitly refer to the exponent and significand fields of the numbers to be added; as noted above, these quantities are very expensive to obtain compared with the cost of a few additions and comparisons. Moreover, Kahan notes that the actual cost of these algorithms in practice is usually far less than the worst case bound, so we are not likely to gain anything by continuing to look for asymptotically cheaper distillation methods.

5. Multiplication

To compute the exact product of two t-digit expansions, we must guarantee that no significant digits are lost when the product of two components is computed in t-digit arithmetic. We accomplish this by splitting each component into a sum of either two or three numbers, each with fewer nonzero digits than the original components. We may then multiply these split components with no error, finally adding all the partial products and renormalizing to obtain a t-digit expansion. The method of splitting components was first proposed by Dekker [4], and we shall rely on the proof given by Linnainmaa [14]. To clarify the statements and proofs, we define the number of *leading nonzero digits* of a t-digit number x to be the smallest integer d such that $x = m\beta^k$ with $\beta^{d-1} \leq |m| < \beta^d$, or zero if $x = 0$.

PROPOSITION 4.1: Let x be a t-digit floating point number. For $1 < k < t$, define $a_k = \beta^{t-k} + 1$. Provided the floating point arithmetic is faithful, the following algorithm computes t-digit numbers x' and x'' such that $x = x' + x''$, x' has at most k leading nonzero digits, and x'' has at most $t - k + 1$ leading nonzero digits.

ALGORITHM 4.1 (Splitting of a t-digit number)

procedure split(x, k)
begin
$y := \mathrm{fl}(a_k \times x)$, $z := \mathrm{fl}(y - x)$
$x' := \mathrm{fl}(y - z)$, $x'' := \mathrm{fl}(x - x')$
return x', x''
end

PROOF: This is Theorem 1 in [14]. Linnainmaa actually states the theorem with an additional hypothesis which guarantees that in fact x'' has at most $t - k$ leading nonzero digits; in a remark following his proof, however, he shows that even if the additional hypothesis is not satisfied, x' will be computed with $k - 1$ leading nonzero digits and x'' will have $t - k + 1$ leading nonzero digits. ∎

We are now ready to give the multiplication algorithm. Note that we choose to split each component of the first expansion into two parts and each component of the second expansion into three parts. Dekker [4] shows that with binary correctly rounding arithmetic it suffices to split each factor into just two parts; Linnainmaa [14] gives other criteria under which splitting into two parts is sufficient. Consequently, the following algorithm can be improved in many cases. Note that we also do not sum the partial products into one net accumulator but instead separate them into groups (denoted $a^{(1)}, \ldots, a^{(6)}$) which are guaranteed to overlap by at most two digits so that they may be renormalized without first being processed through the add algorithm. We then accumulate these renormalized expansions into a smaller accumulator (b) which in turn is added to the overall product accumulator. This nesting of additions significantly reduces the coefficient of the largest order term in the cost estimate.

PROPOSITION 4.2: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components. Assume the arithmetic is faithful and that $t \geq 7$, and set $k_2 = \lfloor t/2 \rfloor + 1$ and $k_3 = \lfloor t/3 \rfloor + 1$. Then the following algorithm computes the product xy expressed as a t-digit expansion with at most $2nm$ components. The algorithm requires at most $98n^2m + 1049nm + 8m - 446n$ t-digit floating point operations.

ALGORITHM 4.2: (Multiplication)

procedure multiply($n, x_1, \ldots, x_n, m, y_1, \ldots, y_m$)
begin
for $i := 1, 2, \ldots, n$

$(x_i', x_i'') := \text{split}(x_i, k_2)$
for $i := 1, 2, \ldots, m$
$(y_i', z) := \text{split}(y_i, k_3)$
$(y_i'', y_i''') := \text{split}(z, k_3)$
$p_1 := 0,\ k := 1$
for $i := 1, 2, \ldots, n$
for $j := 1, 2, \ldots, m$
$a_j^{(1)} := \text{fl}(x_i' \times y_j')$
$a_j^{(2)} := \text{fl}(x_i' \times y_j'')$
$a_j^{(3)} := \text{fl}(x_i' \times y_j''')$
$a_j^{(4)} := \text{fl}(x_i'' \times y_j')$
$a_j^{(5)} := \text{fl}(x_i'' \times y_j'')$
$a_j^{(6)} := \text{fl}(x_i'' \times y_j''')$
$(j, b_1, \ldots, b_j) := \text{renorm}(m, a_1^{(1)}, \ldots, a_m^{(1)})$
$(l, c_1, \ldots, c_l) := \text{renorm}(m, a_1^{(2)}, \ldots, a_m^{(2)})$
$(j, b_1, \ldots, b_j) := \text{renorm}(\text{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \text{renorm}(m, a_1^{(3)}, \ldots, a_m^{(3)})$
$(j, b_1, \ldots, b_j) := \text{renorm}(\text{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \text{renorm}(m, a_1^{(4)}, \ldots, a_m^{(4)})$
$(j, b_1, \ldots, b_j) := \text{renorm}(\text{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \text{renorm}(m, a_1^{(5)}, \ldots, a_m^{(5)})$
$(j, b_1, \ldots, b_j) := \text{renorm}(\text{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(l, c_1, \ldots, c_l) := \text{renorm}(m, a_1^{(6)}, \ldots, a_m^{(6)})$
$(j, b_1, \ldots, b_j) := \text{renorm}(\text{add}(l, c_1, \ldots, c_l, j, b_1, \ldots, b_j))$
$(k, p_1, \ldots, p_k) := \text{renorm}(\text{add}(j, b_1, \ldots, b_j, k, p_1, \ldots, p_k))$
return $k, p_1, \ldots, p_k$
end

Proof: Note that for each i both x_i', x_i'' have at most $\lfloor t/2 \rfloor + 1$ leading nonzero digits, and for each j, y_j', y_j'', y_j''' have at most $\lfloor t/3 \rfloor + 1$ leading nonzero digits. Since $t \geq 7$, each of the six products in the innermost for loop is computed exactly. By the non-overlapping condition for expansions together with the bounds on the number of leading nonzero digits of split components, we see that the six sequences $a_j^{(1)}, \ldots, a_j^{(6)}$, $j = 1, \ldots, m$ computed in the inner loop overlap by at most two digits. Hence the renormalization and addition procedures in the outer loop succeed without error, so the output satisfies $p = xy$. Note that for each i the sums accumulated in b form expansions with at most $2m$ components. Therefore p never exceeds $2nm$ components. A straightforward operation count completes the proof. ∎

The above multiplication algorithm is based on the classical algorithm of multiplying digits pairwise; hence, the number of multiplications is proportional to the product of the numbers of components in each expansion. In particular, if both expansions have n components, the number of basic multiply-and-add steps is $O(n^2)$. Algorithms which multiply n-digit numbers in fewer than $O(n \log^2 n)$ steps are known (see Knuth [10]), but we have elected not to use them for several reasons. Specifically, most of these algorithms improve upon the classical method only for very large n, while in practice we would expect to apply the above algorithm only to expansions with relatively few components (e.g., at most 39 for a typical implementation of binary floating point arithmetic using 64 bit storage; see the remarks on underflow in section 8 below). In addition, some of these algorithms are based on integer arithmetic, so that the input is assumed to consist of a contiguous string of n digits representing each multiplicand. Since we are given input in the form of t-digit expansions, the digits of which need not be contiguous, we cannot always benefit from the application of the techniques suggested.

Notice that although the total number of multiplications is only proportional to nm, the total cost is proportional to n^2m. This extra factor of n in the total cost arises from the cost of adding the partial products. Using slightly more intermediate storage, we can rearrange the order of the additions in the form of a binary tree, much like the arrangement in algorithm 3 above, and reduce this term to $nm \log n$. The remark following algorithm 3 suggests, however, that reducing the cost to $O(nm)$ is not practical.

6. Division

Our division algorithm is analogous to the usual method of long division. In that algorithm, we take the quotient of the most significant digits of the current dividend and divisor as a guess for the next digit of the overall quotient. If the product of this guess with the divisor is larger than the dividend, however, we must choose a smaller digit. For our algorithm, however, since we do not require the components of an expansion to have the same sign, we do not need to adjust the guess in this way. We simply take the quotient of the most significant components (as computed by t-digit arithmetic) as the next term of the quotient, and we show that once a sufficient number of terms have been computed, we can renormalize to express the quotient as an expansion.

Proposition 5: Let $x = \sum_{i=1}^{n} x_i$ be a t-digit expansion with n components and $y = \sum_{i=1}^{m} y_i$ a t-digit expansion with m components, and suppose $y \neq 0$. Assume the arithmetic is faithful and $t \geq 9$; for $d > 0$ let $k = \lfloor (t-4)d/t \rfloor$. Then the following algorithm computes a t-digit expansion $q = \sum q_i$ with at most d components such that $|q - x/y| < \beta^{1-k}|x/y|$. The algorithm requires at most $98(d-1)^2 m + (d-1)(1255m + 49n - 479) + 7$ t-digit floating point operations.

ALGORITHM 5: (Division)

```
procedure divide( d, n, x_1, ..., x_n, m, y_1, ..., y_m )
begin
j^(1) := n
for i := 1, 2, ..., n
    e_i^(1) := x_i
for i := 1, 2, ..., d
    q_i := fl(e_1^(i) / y_1)
    if i < d
        ( k, f_1, ..., f_k ) := multiply( 1, q_i, m, y_1, ..., y_m )
        for l := 1, 2, ..., k
            f_l := -f_l
        ( j^(i+1), e_1^(i+1), ... ) := renorm( add( k, f_1, ..., f_k, j^(i), e_1^(i), ... ) )
( k, s_1, ..., s_k ) := renorm( i, q_1, ..., q_i )
return k, s_1, ..., s_k
end
```

PROOF: The result is trivial if $x = 0$; for $x \neq 0$ it suffices to show that the sequence $q_1, \ldots, q_i$ computed in the main loop overlaps by at most four digits. Proceeding by induction on i, let q_i be the quantity computed in the i-th iteration of the loop for some i and write $q_i = p\beta^u$ with $\beta^{t-1} \leq |p| < \beta^t$. Note that $q_i = (e_1^{(i)}/y_1)(1+\epsilon)$, $f = q_i y = q_i y_1 (1+\delta)$ and $e_1^{(i)} = e^{(i)}(1+\eta)$ where $|\epsilon|, |\delta|, |\eta| < \beta^{1-t}$. Hence writing $\alpha = 1 - (1+\epsilon)(1+\delta)(1+\eta)$ we have $e^{(i+1)} = e^{(i)} - f = e^{(i)}\alpha$ and $|\alpha| < \beta^{3-t}$. Similarly, $e_1^{(i+1)} = e^{(i+1)}(1+\mu) = e^{(i)}\alpha(1+\mu) = e_1^{(i)}\alpha(1+\mu)(1+\eta)^{-1}$ and hence $q_{i+1} = q_i\alpha(1+\mu)(1+\nu)(1+\eta)^{-1}(1+\epsilon)^{-1}$ with $|\mu|, |\nu| < \beta^{1-t}$. Consequently, $|q_{i+1}| < |q_i|\beta^{4-t} < \beta^{u+4}$. Since $t \geq 9$, the induction hypothesis implies that $q_1, \ldots, q_{i-1}$ are multiples of $\beta^{u+t-4} > \beta^{u+4}$ and the proof is complete, save for an easy operation count. ∎

7. Example: Intersecting a Line Segment and a Line

We illustrate the application of our algorithms to a well-studied problem in computational geometry, that of computing the intersection of a line segment and a line. For the purpose of this section, we assume that we have an arithmetic conforming to the IEEE Standard for Binary Floating Point Arithmetic. In particular, the arithmetic is binary and correctly rounding, so the simplest form of sum_err may be used throughout; in addition, we assume that two different precisions are available, corresponding to two different values of t: *single precision*, for which the precision is t_1 ($t_1 = 24$ in the IEEE standard), and *double precision*, with a precision t_2 (the standard specifies $t_2 = 53$). The arithmetic includes operations between single precision operands yielding a single precision result, operations between double precision operands yielding a double precision result, and two conversion operations: one which converts a single precision number to double precision (exactly), and one which correctly rounds a double precision number to single precision provided no overflow or underflow occurs. Most importantly, we suppose that double precision, as the name implies, carries at least twice as many digits as single precision, i.e., $t_2 \geq 2t_1$; this assumption guarantees that we can multiply single precision numbers exactly by converting to and multiplying with double precision.

The general problem is this: given the coordinates of four points in the plane, P_1, P_2, P_3, and P_4, decide whether the line segment P_1P_2 intersects the line containing P_3 and P_4, and if so, compute the coordinates of the intersection point. This problem arises in many computer graphics and computer-aided design applications; numerous authors have recognized both the importance and difficulty of solving the problem accurately (see [6, 15, 16, 18] and references therein). Milenkovic [16], in particular, has shown that if the single and double precision arithmetics satisfy $t_2 \geq 2t_1+1$ then line and line segment intersection calculations may be computed to full accuracy provided lines and line segments are defined by the coefficients of line equations given as single precision numbers. Unfortunately, his approach requires that line segments defined by endpoint coordinates be perturbed and replaced by new line segments defined by line equation coefficients; he apparently overlooks the fact that perturbing two nearly parallel lines may move their intersection point a large distance. Another practical drawback of Milenkovic's method stems from the fact that intersections are characterized exactly, but each intersection point, once computed, is rounded to single precision, so that subsequent tests fail to indicate that the computed intersection point actually belongs to the lines determining the intersection. Milenkovic sidesteps this problem by arguing that any efficient algorithm would not make such a test because it would be redundant (i.e., the point has already been found to be the intersection of the two lines, so we need not test for its inclusion in either line). In practice, however, many otherwise simple applications of line and segment intersection calculations become much more complicated when non-redundancy is required, due to the overhead of maintaining all accumulated topological knowledge.

We propose instead to compute line and line segment intersections to full accuracy—i.e., correct to single precision—directly from endpoint coordinates using the algorithms presented above as necessary. Unlike Milenkovic's method, our algorithm may easily be modified to allow approximate endpoint intersections to be recognized; in this way, we obtain a consistent set of calculations and no longer need to assume non-redundancy. Ottman, et al, [18] propose essentially the same approach for the intersection problem and other geometric problems. In fact, they show how to compute intersection points to full accuracy using only single precision arithmetic and a means of computing inner products to full precision. Their method, however, relies on interval arithmetic, which in turn requires directed roundings, to obtain the final result to full accuracy. In contrast, our method uses only the faithfulness of the arithmetic and could easily be implemented entirely in single precision by using the multiplication and division algorithms of sections 5 and 6. Before presenting the algorithm, we formulate a concise statement of the problem.

PROBLEM: Given single precision numbers (x_1, y_1), (x_2, y_2), (x_3, y_3), and (x_4, y_4). Decide whether the line segment $(x_1, y_1)(x_2, y_2)$ intersects the line determined by (x_3, y_3) and (x_4, y_4) at a unique point, and if so, compute the coordinates (x, y) of the intersection point accurate to single precision. (That is, if the exact intersection point is (x', y'), then return a point (x, y) such that x is the nearest single precision number to x' and y is the nearest single precision number to y'.)

We first devise a straightforward but somewhat costly algorithm which is easily seen to solve the problem. Writing the intersection problem as a system of two linear equations, we see that the line segment intersects the line at a unique point if and only if $d \neq 0$, where $d = (y_4 - y_3)(x_1 - x_2) + (x_3 - x_4)(y_1 - y_2)$. In this case, the coordinates of the intersection point are given by

$$x = \frac{1}{d}\left[(x_1 - x_2)t + (x_3 - x_4)s\right]$$

$$y = \frac{1}{d}\left[(y_1 - y_2)t + (y_3 - y_4)s\right]$$

where $s = (x_2 y_1 - x_1 y_2)$ and $t = (x_3 y_4 - x_4 y_3)$. We can compute d by rewriting the previous expression as a sum of products, multiplying each term using double precision, then distilling the terms to obtain an expansion for d. (This is precisely the manner in which Ottman, et al, use the accurate inner product evaluation.) Then d vanishes if and only if the first component of the expansion vanishes, and otherwise this component is accurate to double precision. Similarly, we can express the numerators of the expressions for x and y as sums of products, then multiply these terms (although since each term has degree three, we need to use the Dekker/Linnainmaa splitting method), distill, and use the leading components, which are again accurate to double precision. Taking the quotient of these leading components with the leading component of d gives coordinates which are accurate to near double precision, and at any rate sufficiently accurate to round to the correct single precision quantities, provided no underflow occurs. We formalize these ideas in the following proposition; we omit the straightforward proof. (Since the number of terms being distilled is never larger than sixteen, we have assumed for the operation count that Pichat's distillation algorithm is used.)

PROPOSITION: The following algorithm solves the above problem using at most 3301 IEEE 754-compatible double precision floating point arithmetic operations including precision conversions. Moreover, no computation overflows, and the only computation which can underflow is the final conversion to single precision. (Here fl denotes double precision computation.)

ALGORITHM: (Intersect line and segment, version 1)

procedure intersect1($x_1, y_1, \ldots, x_4, y_4$)
begin
convert $x_1, y_1, \ldots, x_4, y_4$ to double precision numbers $x'_1, y'_1, \ldots, x'_4, y'_4$
$e_1 := \mathrm{fl}(x'_1 \times y'_4)$, $e_2 := \mathrm{fl}(-x'_1 \times y'_3)$
$e_3 := \mathrm{fl}(-x'_2 \times y'_4)$, $e_4 := \mathrm{fl}(x'_2 \times y'_3)$
$e_5 := \mathrm{fl}(x'_3 \times y'_1)$, $e_6 := \mathrm{fl}(-x'_3 \times y'_2)$
$e_7 := \mathrm{fl}(-x'_4 \times y'_1)$, $e_8 := \mathrm{fl}(x'_4 \times y'_2)$
$(n, d_1, \ldots, d_n) :=$ distill($8, e_1, \ldots, e_8$)
if $d_1 = 0$
    return "No unique intersection"
else
    $(e'_1, e''_1) :=$ split(e_1, t_1), ... , $(e'_8, e''_8) :=$ split(e_8, t_1)
    $b_1 := \mathrm{fl}(e'_1 \times x'_3)$, ... ,$b_{16} := \mathrm{fl}(e''_8 \times x'_1)$
    $(j, b_1, \ldots, b_j) :=$ distill($16, b_1, \ldots, b_{16}$)
    $c_1 := \mathrm{fl}(e'_5 \times y'_4)$, ... ,$c_{16} := \mathrm{fl}(e''_1 \times y'_2)$
    $(k, c_1, \ldots, c_k) :=$ distill($16, c_1, \ldots, c_{16}$)
    $x' := \mathrm{fl}(b_1/d_1)$, $y' := \mathrm{fl}(c_1/d_1)$
    convert x', y' to single precision numbers x, y
    return x, y
end

The preceding algorithm is easy to deduce, but seems rather costly. Can we do better? For a global algorithm, the answer seems to be "no", since we must avoid destructive cancellation in computing sums. Nevertheless, we can rearrange the computation slightly to guarantee that only two sums can incur destructive cancellation, and then only rarely, by proceeding as follows. Note that we may write the coordinates of the intersection point as $x = x_1 + a(x_2 - x_1)$ and $y = y_1 + a(y_2 - y_1)$ where $a = u/(u - v)$, $u = (y_4 - y_3)(x_1 - x_3) + (x_3 - x_4)(y_1 - y_3)$, and $v = (y_4 - y_3)(x_2 - x_3) + (x_3 - x_4)(y_2 - y_3)$. In terms of u and v, we have the following possibilities: if $uv > 0$, the line segment does not intersect the line; if $u = 0$ and $v = 0$, the line segment coincides with the line; if $u = 0$ but $v \neq 0$, the line segment intersects the line at P_1, and similarly if $u \neq 0$ but $v = 0$; finally, if $uv < 0$, the line segment intersects the line at some point between P_1 and P_2.

Suppose we compute u and v accurate to double precision, by means described above. We can test for each of the aforementioned cases, and we need only perform additional computations if $uv < 0$. In this case, however, no cancellation occurs when we compute $u - v$, so a can be computed to near double precision. Likewise, we can compute $x_2 - x_1$ and $y_2 - y_1$ to near double precision, and therefore the products $a(x_2 - x_1)$ and $a(y_2 - y_1)$ may be computed to near double precision. Specifically, each of these quantities will be accurate to all but the last two significant bits. Thus if neither sum $x_1 + a(x_2 - x_1)$ and $y_1 + a(y_2 - y_1)$ cancels in more than $t_2 - 2 - t_1$ leading bits, then both will be sufficiently accurate to round to the correct single precision values. Moreover, we can test whether this much cancellation has occurred, and if so, we can always resort to the lengthy algorithm presented above. To summarize, we state our results (again without proof).

PROPOSITION: Let $c = 2^{t_2 - 2 - t_1}$. The following algorithm solves the above problem using at most 721 IEEE 754-compatible double precision floating point operations including conversions, *provided the condition in the last if statement fails*. Moreover, no computation overflows, and the only computation which can underflow is the final conversion to single precision. (Again, fl denotes computation in double precision.)

ALGORITHM: (Intersect line and segment, version 2)

procedure intersect2($x_1, y_1, \ldots, x_4, y_4$)
begin
convert $x_1, y_1, \ldots, x_4, y_4$ to double precision numbers $x'_1, y'_1, \ldots, x'_4, y'_4$
$u_1 := \mathrm{fl}(y'_4 \times x'_1), \ldots, u_8 = \mathrm{fl}(x'_4 \times y'_3)$
$(j, u_1, \ldots, u_j) := \mathrm{distill}(8, u_1, \ldots, u_8)$
$v_1 := \mathrm{fl}(y'_4 \times x'_1), \ldots, v_8 = \mathrm{fl}(x'_4 \times y'_3)$
$(k, v_1, \ldots, v_k) := \mathrm{distill}(8, v_1, \ldots, v_8)$
if $u_1 = 0$
    if $v_1 = 0$
        return "Intersection not unique"
    else
        return x_1, y_1
else if $v_1 = 0$
    return x_2, y_2
else if ($u_1 < 0$ and $v_1 < 0$) or ($u_1 > 0$ and $v_1 > 0$)
    return "No intersection"
$a := \mathrm{fl}(u_1/\mathrm{fl}(u_1 - v_1))$
$x' := \mathrm{fl}(x'_1 + \mathrm{fl}(a \times \mathrm{fl}(x'_2 - x'_1)))$, $y' := \mathrm{fl}(y'_1 + \mathrm{fl}(a \times \mathrm{fl}(y'_2 - y'_1)))$
if $|x'| \le \mathrm{fl}(c \times |x'_1|)$ or $|y'| \le \mathrm{fl}(c \times |y'_1|)$
    $(x, y) := \mathrm{intersect1}(x_1, y_1, \ldots, x_4, y_4)$

```
else
      convert x', y' to single precision numbers x, y
return x, y
end
```

The last if test fails only when either the x or y coordinate of the intersection point is much closer to zero than x_1, respectively y_1. We cannot expect to give a meaningful estimate of the probability with which the test fails, but we can say loosely that it will happen "only rarely". Even then, we simply resort to the longer computation given as version 1, and here we could exercise greater care and reuse some of the quantities already computed in version 2, so that the overall cost will be slightly less than the sum of the costs of each version alone. (In this respect, our approach is similar to that suggested by Dobkin and Silver [6].) Moreover, the most pessimistic quantities in the cost estimates arise from the distillation procedure, which, as Kahan [9] notes, often takes far less time than we can predict. Therefore, we suggest that the preceding algorithm is not only provably accurate and robust, but usually very efficient as well.

8. Further Remarks

We have exhibited algorithms which perform exact addition and multiplication and arbitrarily precise division using only fixed precision floating point arithmetic operations by expressing extended precision numbers as unevaluated sums of t-digit floating point numbers. The algorithms are based on a simple property possessed by most "reasonable" implementations of floating point arithmetic. As an application of these algorithms, we have given a provably robust, accurate, and reasonably efficient algorithm for computing the intersection of a line and a line segment. Unlike many algorithms which are *backwards stable* in that they actually compute the intersection of a nearby line and line segment, our algorithm is *forwards stable*: it computes the point nearest the exact intersection of the given line and segment.

We express the cost of our algorithms as the number of t-digit floating point arithmetic operations required as a function of the number of components of the input expansions. Note that a typical number in R_β does not have a unique t-digit expansion; in fact, even the number of components is not uniquely determined. Since the cost of each operation depends on the number of components, we could in principle demand that we always compute *minimal* t-digit expansions. We choose not to do so, however, since this would require the development of an algorithm which finds a minimal expansion given an arbitrary one, and such algorithms are difficult to formulate in a machine-independent way. Instead we only require that an expansion satisfy the non-overlapping condition; i.e., that significant digits of successive components do not overlap. This requirement incurs additional cost in the form of the renormalization steps following each operation, but it provides a convenient form for subsequent rounding and usually leads to "nearly minimal" expansions. We conjecture that for many applications, the cost of producing minimal expansions outweighs the cost of working with nearly minimal ones. (In fact, for some applications we may save time by renormalizing even less frequently; our algorithms could easily be made more efficient in this respect, although some of the proofs would need to be refined.)

In designing our algorithms, we have tacitly assumed that the exponent range of the arithmetic is unlimited, so that no overflow or underflow occurs. In fact, although it is possible for our distillation and multiplication algorithms to overflow when the exact results lie within representable range, such overflow can occur only when the inputs are already very close to the overflow threshhold and is therefore not likely. On the other hand, since the trailing components of expansions quickly become very small, underflow imposes a certain limitation on the accuracy of these algorithms. (For example, IEEE standard double precision with a 64 bit storage format contains eleven bits of exponent and 53 bits of fraction. Hence the net exponent range is $2^{11} = 2048$, so the maximum number of representable components in an expansion is $\lceil 2048/53 \rceil = 39$. Exact calculations can easily exceed this limit after several operations.) Nevertheless, we believe that in most applications, intermediate results may be rounded or rescaled so that underflow does not pose a serious problem. Even when underflow cannot be avoided, Demmel [5] points out that the cumulative effect of the resulting errors can often be estimated by extending the error analysis to include an underflow term whose magnitude is bounded by the underflow threshhold, if underflow is gradual, or by the ratio of this threshhold to the unit roundoff otherwise.

In conclusion we note that the techniques we have proposed for calculating to arbitrary precision can be used equally well to simulate either floating point or fixed point arithmetic. Specifically, if after each calculation we truncate the resulting expansion to a fixed number of components, then the relative error of each intermediate result will be bounded by a corresponding constant, as is true of single precision floating point arithmetic. Alternatively, if we drop only trailing components whose magnitude is below a given threshhold (as with underflow), then we can instead bound the absolute error of each operation, thereby emulating fixed point arithmetic. More importantly, in either case we may adjust the precision dynamically, calculating certain intermediate results to very high precision and others to lower precision, to obtain the desired accuracy in the solution; the cost of the computation, not the precision of the arithmetic, determines the size of acceptable errors. We should then rephrase the traditional question of roundoff error analysis, "How accurate is a computed result if we calculate each intermediate quantity using fixed precision arithmetic?", as the more natural question, "What is the cheapest way to compute a result to a specified accuracy?" We believe further study in this direction will provide both practical and theoretically useful results.

Acknowledgements

I am deeply indebted to Prof. W. Kahan, who has helped me to understand the significance of this work. I also wish to thank the referees for many helpful comments and suggestions.

References

[1] Bohlender, G., Floating-Point Computation of Functions with Maximum Accuracy, *IEEE Trans. Comput.* **C-26** (1977), 621–632.

[2] Bohlender, G., What Do We Need Beyond IEEE Arithmetic?, in Ullrich, C. (Ed.), *Computer Arithmetic and Self-Validating Numerical Methods*, Academic Press, Boston, 1990.

[3] Brent, R., A Fortran Multiple Precision Arithmetic Package, *ACM Trans. Math. Soft.* **4** (1978), 57–70.

[4] Dekker, T., A Floating-Point Technique for Extending the Available Precision, *Numer. Math.* **18** (1971), 224–242.

[5] Demmel, J., Underflow and the Reliability of Numerical Software, *SIAM J. Sci. Stat. Comput.* **5** (1984), 887–919.

[6] Dobkin, D., and D. Silver, Recipes for Geometry and Numerical Analysis, in *Proc. Fourth Annual Symposium on Computational Geometry*, Association for Computing Machinery, New York, New York, 1988.

[7] Kahan, W., Further Remarks on Reducing Truncation Errors, *Comm. ACM* **8** (1965), 40.

[8] ——, A Survey of Error Analysis, in Freeman, C. V. (Ed.), *Proc. IFIP Cong. 1971*, vol. 2, North-Holland, Amsterdam, 1972.

[9] ——, Paradoxes in Concepts of Accuracy, lecture notes from Joint Seminar on Issues and Directions in Scientific Computation, U. C. Berkeley, 1989.

[10] Knuth, D., *The Art of Computer Programming*, vol. 2 (2nd ed.), Addison-Wesley, Reading, Mass., 1981.

[11] Kulisch, U., and W. Miranker, *Computer Arithmetic in Theory and Practice*, Academic Press, New York, New York, 1981.

[12] Leuprecht, H., and W. Oberaigner, Parallel Algorithms for the Rounding-Exact Summation of Floating-Point Numbers, *Computing* **28** (1982), 89–104.

[13] Linnainmaa, S., Analysis of Some Known Methods of Improving the Accuracy of Floating-Point Sums, *BIT* **14** (1974), 167–202.

[14] ——, Software for Doubled-Precision Floating-Point Computations, *ACM Trans. Math. Soft.* **7** (1981), 272–283.

[15] Milenkovic, V., Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic, *Artificial Intelligence* **37** (1988), 377–401.

[16] ——, Double Precision Geometry: A General Technique for Calculating Line and Segment Intersections Using Rounded Arithmetic, in *Proc. Thirtieth Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Los Alamitos, Calif., 1989.

[17] Møller, O., Quasi Double Precision in Floating-Point Addition, *BIT* **5** (1965), 37–50.

[18] Ottman, T., G. Thiemt, and C. Ullrich, Numerical Stability of Simple Geometric Algorithms in the Plane, in Børger, E. (Ed.), *Computation Theory and Logic*, Springer-Verlag, Berlin, 1987.

[19] Pichat, M., Correction d'une Somme en Arithmetique à Virgule Flottante, *Numer. Math.* **19** (1972), 400–406.

[20] Sterbenz, P., *Floating-Point Computation*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

[21] Wilkinson, J., *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey, 1964.